Diseño e implementación del soporte para FPGA en OmpSs-2

Especialidad Ingeniería de Computadores

Autor
Ruben Cano Díaz

Director
Daniel Jimenez Gonzalez [AC]

Codirector
Xavier Martorell Bofill [AC]

3 de junio 2018

Agradezco a mi tutor del TFG por todo el soporte proporcionado
A mis queridos compañeros de VGAFIB
y a Noelia por tener que aguantarme día tras día.

Índice

Abstract

Español

En este trabajo nos centraremos en los cambios necesarios que hay que hacer al *runtime* Nanos6 para poder ejecutar tareas FPGA. Para ello, analizaremos cada una de las partes del *runtime* encargadas de la ejecución de tareas, y, tras ello, para cada una de las partes, explicaremos qué hay que modificar y añadir para que pueda funcionar nuestra implementación.

Català

En aquest treball ens centrarem en els canvis necessaris que cal fer al runtime Nanos6 per poder executar tasques FPGA. Per a això, analitzarem cadascuna de les parts del runtime encarregades de l'execució de tasques, i, per a cadascuna de les parts, explicarem què cal modificar i afegir perquè pugui funcionar la nostra implementació.

English

In this paper we will focus on the necessary changes that must be made to Nanos6 runtime in order to execute FPGA tasks. To do this, we will analyze each of the parts of the runtime responsible for the execution of tasks, and after that, for each of the parts, we will explain what needs to be modified or added in order to get our implementation working.

1 Introducción

Este Trabajo Final de Grado (TFG) es un proyecto del Grado en Ingeniería Informática, especializado en Ingeniería de Computadores, ofrecido por la Facultad de Informática de Barcelona, perteneciente a la Universidad Politécnica de Cataluña.

Para la realización del proyecto, se cuenta con la colaboración del Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC - CNS)[1], pues el propósito de éste trabajo se centra en la implementación del soporte para devices FPGA en el *runtime* **Nanos6**[2], utilizado en el modelo de programación heterogénea **OmpSs-2**[3] desarrollado en el BSC.

OmpSs-2 es un nuevo modelo de programación paralela y heterogénea que extiende el modelo de ***tasks*** de OmpSs[4] y OpenMP[5]. OmpSs-2 se ha desarrollado para solventar los problemas de diseño de la primera versión (OmpSs) que podían afectar al rendimiento de manera negativa.

El modelo de programación OmpSs-2 es formado por la unión entre el compilador *Source to Source* **Mercurium**[6] y Nanos6. Mercurium se encargará, entre otras cosas, de hacer las llamadas al *runtime* Nanos6, encargado de la gestión de los dispositivos y de la ejecución de los programas en ellos.

[1] "BSC-CNS | Barcelona Supercomputing Center - Centro Nacional de ...." https://www.bsc.es/. Se consultó el 5 mar.. 2018.
[2] "GitHub - bsc-pm/nanos6: Nanos6 is a runtime that implements the ...." 7 nov.. 2017, https://github.com/bsc-pm/nanos6. Se consultó el 5 mar.. 2018.
[3] "Release of OmpSs-2 programming model | BSC-CNS." 13 nov.. 2017, https://www.bsc.es/news/bsc-news/release-ompss-2-programming-model. Se consultó el 5 mar.. 2018.
[4] "The OmpSs Programming Model | Programming Models @ BSC." https://pm.bsc.es/ompss. Se consultó el 5 mar.. 2018.
[5] "OpenMP Application Programming Interface." 2 mar.. 2017, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. Se consultó el 5 mar.. 2018.
[6] "GitHub - bsc-pm/mcxx: Mercurium is a C/C++/Fortran source-to-source ...." https://github.com/bsc-pm/mcxx. Se consultó el 5 mar.. 2018.

1.1 Contexto

En los últimos años hemos visto que la computación de alto rendimiento ha progresado desde la utilización de procesadores para la realización del cálculo, a la utilización de diferentes dispositivos con capacidad de computación paralela superior, o con diferentes características.

La introducción de estos nuevos dispositivos a su utilización en **HPC**[7] supone una serie de retos que hay que solventar, y, para ello, se crean los modelos de programación.

En el BSC y otras entidades de investigación, se desarrollan modelos de programación con el objetivo de ofrecer a los programadores la capacidad de centrarse en el problema a solventar, haciendo que la programación sobre plataformas con diferentes dispositivos de cálculo o **heterogéneas** sea más sencilla, facilitando así el uso de estos dispositivos tanto a expertos como a noveles.

1.2 Actores implicados

- **Desarrolladores:** La implementación del *device* FPGA es responsabilidad del alumno implicado en el proyecto. A pesar de ello, OmpSs-2 un modelo de programación en desarrollo, el cual debe mantener una cierta coherencia entre las implementaciones de diferentes devices. Debido a ello se trabajará en algunas ocasiones junto con el actual responsable de la implementación para el *device* GPU[8].

- **Director y codirector:** Tanto el director como codirector se encargan de apoyar, guiar y supervisar al desarrollador en las tareas en las que necesite ayuda, así como de fijar y marcar los objetivos y llevar un seguimiento del trabajo realizado.

- **Barcelona Supercomputing Center:** El BSC se implica ofreciendo los recursos *software* y *hardware* necesarios para el desarrollo del proyecto, entre ellos el acceso a el control de versiones utilizado internamente por el BSC y a las placas de desarrollo para FPGAs sobre las que deberá funcionar la implementación.

- **Beneficiarios:** El beneficiario principal será el Barcelona Supercomputing Center, debido a que una funcionalidad existente en OmpSs, pasará a estar disponible en

[7] "What is high performance computing? - insideHPC." https://insidehpc.com/hpc-basic-training/what-is-hpc/. Se consultó el 5 mar.. 2018.
[8] "Design and Development of support for GPU Unified ... - UPCommons." 31 oct.. 2017, https://upcommons.upc.edu/bitstream/handle/2117/112437/126955.pdf. Se consultó el 5 mar.. 2018.

OmpSs-2. A su vez, cualquier personaje interesado en utilizar OmpSs-2 como modelo de programación para sus proyectos, verá ampliadas sus posibilidades con la inclusión del soporte para FPGA, haciendo que se pueda utilizar OmpSs-2 en nuevas plataformas en las que hasta ahora igual no era lo suficientemente interesante como para justificar su uso.

1.3 FPGA

Se le llama FPGA o *Field-programmable gate array* a un dispositivo programable que contiene bloques hardware de lógica cuya interconexión y funcionalidad puede ser configurada.

En los últimos años, **Intel**[9] y otras compañías han empezado a apostar por el uso de FPGAs para High Performance Computing o **HPC**.

Una de las características que las hace entrar en la competición de la alta computación es su masivo paralelismo y la capacidad de adaptar su configuración al problema que se está resolviendo.

De esta forma, se puede resolver problemas más rápidamente que un procesador convencional, además de en muchos casos, resolverlos en un tiempo determinístico (Es decir, que se puede saber cuánto tardará con exactitud a partir de los valores de entrada).

2 Estado del arte

En este proyecto se pretende implementar el soporte para FPGAs en el *runtime* Nanos6, soporte que ya existe en el *runtime* Nanos5.

La implementación para devices diferentes está en pleno desarrollo, en especial la implementación del soporte para GPU utilizando memoria unificada.

Éste proyecto heredará de los avances en el desarrollo de la implementación para GPU e intentará aportar a aquellas partes en las que se pueda considerar útil.

[9] "Intel Launches FPGA Accelerator Aimed at HPC and HPDA ... - Top500." 20 dic.. 2017, https://www.top500.org/news/intel-launches-fpga-accelerator-aimed-at-hpc-and-hpda-applications/. Se consultó el 5 mar.. 2018.

2.1 Modelos de programación OmpSs y OmpSs-2

Un modelo de programación es una forma de expresar en código de alto nivel el paralelismo.

OmpSs y OmpSs-2 son dos modelos de programación desarrollados en el BSC basados en OpenMP+StarSs con el objetivo de extender C, C++ y FORTRAN con nuevas directivas de programación. Permitiendo así la ejecución de programas paralelos.

Además, estos cuentan con la posibilidad de ser ejecutados en plataformas heterogéneas, o lo que es lo mismo, máquinas que contengan más de un dispositivo de cálculo.

Con ese objetivo en mente, el modelo OmpSs se construye a partir del a combinación de dos piezas, el compilador **Mercurium** y el ***runtime***.

2.1.2 Mercurium

El compilador Mercurium es un compilador *Source to Source* que entiende las directivas escritas por el programador, transformando y añadiendo el código necesario para que el programa actúe de la forma esperada por las especificaciones del modelo.

Para ese fin, el compilador añade llamadas al ***runtime***, que se encargará de gestionar los diferentes dispositivos y ejecutar los *threads* del programa, entre otras cosas.

2.1.3 Runtime

El *runtime* es el encargado de implementar las especificaciones del modelo de programación, se encarga de las siguientes tareas:

Asignar memoria: en un dispositivo de memoria, que puede ser RAM, o cualquier otro dispositivo soportado.

Asignar un dispositivo de ejecución: sea CPU, GPU, FPGA o cualquier otro dispositivo,

Scheduling: un Scheduler es el encargado de suministrar el orden a ejecutar las diferentes tasks asignadas a los diferentes dispositivos.

Executor: Un conjunto de threads que, tras recibir una task del Scheduler, proceden a la ejecución de la task asignada, tras su finalización, avisan al gestor de dependencias para que threads bloqueados por las dependencias puedan continuar.

Nanos++

Nanos++ es la versión del *runtime* utilizado en OmpSs, hasta ahora, es la más completa en soporte a dispositivos y funcionalidades. .

En el siguiente ejemplo, podemos ver como sería la programación de una tarea para su ejecución en una fpga utilizando el modelo de programación OmpSs.

```
#pragma omp target device(fpga) copy_deps
#pragma omp task inout([NB*NB]C) in([NB*NB]A, [NB*NB]B)
void matmul_tile(REAL *A, REAL *B, REAL *C)
{
        int i, j, k;
        //implements C <= alpha*A*B + beta*C, with alpha=1.0
         & beta=1.0
        //this is the code that should be done in the FPGA

        for(i=0; i<NB; i++){
                for(k=0; k < NB; k++){
                        REAL tmp = A[i*NB+k];
                        for(j=0; j < NB; j++){
                                C[i*NB+j] += tmp * B[k*NB+j];
                        }
                }
        }
}
```

Figura 1: Implementación de un algoritmo para ser ejecutado en una FPGA

Nanos6

OmpSs-2 es un nuevo modelo de programación basado en OmpSs y sigue utilizando el compilador Mercurium. El *runtime* se ha reescrito desde cero y se desarrolla con la idea de arreglar algunos enfoques en el diseño que podían afectar al rendimiento y extensión del modelo de programación.

Se espera que esta nueva implementación del *runtime* pueda alcanzar y superar en características a Nanos++.

En éste proyecto, ampliaremos las características de Nanos6 añadiendo la posibilidad de tener un *device* **FPGA**. Esta función ya existía en la anterior implementación, pero no se encuentra disponible en Nanos6.

2.2 OpenACC

OpenACC es un modelo de programación estándar para programación paralela, pensado para computadores híbridos CPU/GPU.

Está basado en directivas y, aunque actualmente no soporta el soporte para FPGA, puede que en un futuro lleguemos a ver ésta posibilidad, pues hay entidades que han apostado por ofrecer una sintaxis para su implementación[10].

2.3 OpenMP

OpenMP es un modelo de programación estándar para programación paralela, pensado en un principio para CPUs, pero actualmente también dispone de soporte para GPU

Está basado en directivas, y, aunque actualmente no soporta el soporte para FPGA, puede que en un futuro lleguemos a ver ésta posibilidad, pues hay entidades que han apostado por ofrecer una sintaxis para su implementación.[11] [12]

2.4 OpenCL

OpenCL es un modelo de programación abierto pensado para ejecutar aplicaciones en plataformas heterogéneas. Este modelo de programación

Una de las principales diferencias que tiene OpenCL respecto a los modelos de programación explicados con anterioridad, es que ofrece más control sobre el código que se genera en el host, es decir, tenemos que hacer nosotros las transferencias de datos, llamadas a los kernels o programas directamente utilizando funciones propias de OpenCL.

Esto se puede ver como un punto positivo o negativo, pues aunque tengas más control, la verbosidad del código aumenta, así como su dificultad y las posibilidad de hacer algo mal.

---

[10] "OpenACC to FPGA - Future Technologies Group - Oak Ridge National ...." https://ft.ornl.gov/sites/default/files/IPDPS16_OpenACC2FPGA_PPT.pdf. Se consultó el 6 mar.. 2018.
[11] "OpenMP extensions for FPGA Accelerators." https://ce-publications.et.tudelft.nl/publications/323_openmp_extensions_for_fpga_accelerators.pdf. Se consultó el 6 mar.. 2018.
[12] "Generating Hardware From OpenMP Programs - NUS Computing." https://www-new.comp.nus.edu.sg/~wongwf/papers/fpt06.pdf. Se consultó el 6 mar.. 2018.

```
__attribute__((num_simd_work_items(8)))
mem_stream(__global uint * src, __global uint * dst)
{
      size_t gid = get_global_id(0);
      dst[gid] = src[gid];
}
```

Figura 2: Implementación de un kernel para realizar un memcopy en una fpga utilizando OpenCL

3 Desarrollo del proyecto

3.1 Alcance

Para lograr implementar la funcionalidad que permita la ejecución de programas en una FPGA en OmpSs-2, primero hay que familiarizarse con el diseño y estructura del *runtime* Nanos6.

Por ello, en el estudio inicial del problema, se harán reuniones con diferentes responsables de la implementación del *runtime*, y a su vez, se asistirá a exposiciones internas llevadas a cabo en el BSC con éste objetivo.

Una vez introducido al código y comprendido las diferentes piezas que lo conforman, se trabajará colaborativamente con los encargados de implementar otros devices diferentes para mantener la coherencia en la implementación final, estudiando a su vez la implementación ya existente para la versión del *runtime* Nanos++.

Tras ello, se procederá a la implementación del código extendiendo las capacidades de Nanos6, con el objetivo de lograr ejecutar un programa en una FPGA utilizando este *runtime*.

3.2 Posibles obstáculos y soluciones

Nanos6 es un *runtime* en desarrollo sobre el cual se está acabando de implementar el soporte para *device* GPU, debido a ello, no hay una versión estable del *runtime* a utilizar, las especificaciones de la implementación de los *device* pueden cambiar y pueden surgir *bugs* que provoquen un contratiempo en la realización del proyecto.

En caso de que llegase a ser un problema, se intentaría reducir la complejidad detrás del diseño para lograr el objetivo principal del proyecto, que es ejecutar código en una FPGA desde Nanos6, dejando para otro proyecto o más adelante la adaptación de la implementación a los patrones acordados.

Mercurium, el compilador, no ha sido probado para generar código ni llamadas al *runtime* en modo compatibilidad con Nanos6, aunque aparentemente no debería suponer un problema, podría ser que fuese una fuente de errores o falta de funcionalidad.

En ese caso, se trataría de trabajar con los desarrolladores de Mercurium para solventar el problema, y si no fuese posible, simular las llamadas que haría el compilador al runtime.

3.3 Metodología y rigor

Debido a que el proyecto se lleva en colaboración con el BSC, la forma de encararlo será adaptándose a su metodología de trabajo y utilizando las herramientas colaborativas que ellos mismos utilicen.

Con las características del proyecto actual, una de las metodologías que más parecen encajar en la realización del mismo es la **desarrollo en cascada**[13], la cual consiste en analizar, diseñar y tras ello implementar el sistema que se quiera desarrollar.

La elección de esta metodología tiene una razón de ser, pues uno de los principales motivos para el desarrollo de **Nanos6** es el rediseño del *runtime* anterior para solventar algunas carencias que afectan al rendimiento. Además este sistema suele ayudar a conseguir un código más estructurado y entendible. Esto es debido al previo diseño, mientras que otras metodologías que intentan ofrecer la funcionalidad a costa de la calidad del diseño, podrían poner en riesgo la calidad, estructuración y rendimiento del código resultante.

[13] "Desarrollo en cascada - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Desarrollo_en_cascada. Se consultó el 5 mar.. 2018.

4 Planificación Temporal

Este proyecto se separa en dos partes, la primera, la gestión de proyecto, equivalente a 3 créditos ECTS o un total de 75h, mientras que la segunda es la realización del proyecto, la cual son 15 créditos o el equivalente a 375h. En la planificación temporal, se intentará realizar una proyección del tiempo y orden en el que las diferentes tareas que conforman nuestro proyecto van a ser realizadas.

4.1 Descripción de tareas

4.1.1 Gestión de Proyectos (GEP)

La asignatura de gestión de proyectos, o GEP, ocupa un total de 75h que debemos dividir entre las diferentes actividades evaluativas, las cuales veremos en la siguiente tabla, junto a su justificación en el tiempo asignado.

ENTREGABLE	JUSTIFICACIÓN	HORAS
Definición del alcance y contextualización	Entrega más larga, se empieza de cero, hay que escoger un formato y definir las metas, a la vez que hacer entendible la finalidad del trabajo a realizar.	25
Planificación temporal	Entrega más o menos corta, a partir de ésta, ya hay establecido un modelo de documento.	6
Gestión económica y sostenibilidad	Debido a la naturaleza del proyecto, hay un número reducido de factores en éste ámbito.	6
Presentación preliminar	Siendo que hay que grabar un vídeo, se le ha dado un poco más de tiempo que a las anteriores tareas.	7
Condiciones de especialidad	Debido a la importancia de la justificación de mi especialidad en el desarrollo del proyecto, se ha asignado más tiempo a ésta entrega.	10
Documento final y presentación oral	Se ha asignado un gran número de horas a la última tarea, ya que el entregable final será un punto de partida importante en la realización de la documentación del TFG, así como la necesidad de una presentación oral requiere preparación.	20

Figura 3 Tabla de tareas y dedicación

4.1.2 Familiarización con el entorno

Como es lógico, el primer paso es la familiarización del entorno, indispensable para la realización y correcta implementación del trabajo, en esta fase, además de acudir a reuniones semanales en el BSC, se estudiará la arquitectura del *runtime* Nanos6.

4.1.3 Instalación del entorno de desarrollo

Tras una primera toma de contacto con el entorno de desarrollo, habrá que descargar la versión de Mercurium y Nanos6 desde la que empezará la implementación de nuestro proyecto.

Hay que tener en cuenta que la instalación de un entorno de desarrollo usable no es una tarea trivial, pues requiere de la instalación de las dependencias y de la comprobación de la validez del mismo.

4.1.4 Diseño e implementación

Una vez tenemos listo el entorno de desarrollo, habrá que diseñar el sistema, entre las tareas.

1. **Reconocimiento del *hardware*:** Es necesario reconocer y habilitar el *hardware* necesario al iniciar la aplicación, pues de ello dependerá que se pueda utilizar la funcionalidad añadida.

2. **Integración de la comunicación con el hardware:** Hay que implementar la comunicación en bajo nivel con el hardware, de forma que podamos enviar y transferir información mediante streams, tanto como para enviar las peticiones y los datos como para recibir los valores resultantes.

3. **Generalización de funciones para unificar el diseño:** Aunque diferentes dispositivos tengan diferentes formas de comunicarse, se desarrollará e implementará una interfaz que unifique la interacción con los mismos.

4. **Gestión de memoria:** Hay que generar y asegurar que las estructuras necesarias para controlar las dependencias de los datos son válidas y no pueden comprometer el resultado del programa. También hay que liberar la memoria cuando ya no sea referenciada.

5. **Scheduler:** Hay que introducir un nuevo *scheduler* que se puedan tratar tareas del tipo FPGA, en un principio, se buscará el correcto funcionamiento de una ejecución.

4.1.5 Testeo de funcionalidad

Una vez acabada la implementación, habrá que comprobar su funcionamiento mediante la realización de unos juegos de pruebas, y, a su vez, comparar el rendimiento obtenido respecto a la versión anterior, y, en caso que algo no funcione del todo bien, habrá que descubrir el problema y arreglarlo.

4.1.5 Memoria del trabajo

La memoria del trabajo es una tarea que se realizará constantemente en la medida del desarrollo del mismo. Desde el inicio de GEP se estará trabajando en ella, y se acabará definitivamente una semana antes de la entrega.

4.2 Dependencias entre tareas

En la siguiente tabla se mostrarán las dependencias entre las diferentes tareas que se van a desarrollar durante el trabajo, Todas las tareas dentro de Diseño e implementación dependen de su orden de explicación en su subapartado.

Tarea	**Dependencia**
GEP	-
Familiarización con el entorno	-
Instalación del entorno de desarrollo	Familiarización con el entorno
Diseño e implementación*	Instalación del entorno de desarrollo
Testeo de funcionalidad	Diseño e implementación
Memoria del trabajo	Testeo de funcionalidad

Figura 4: Tabla de dependencias entre tareas

4.3 Gestión de tiempo

En la siguiente tabla, se puede ver una definición de la duración de las tareas, su situación espacial y el riesgo a la que es sujeta cada una de ellas.

Tarea	Inicio	Fin	Riesgo	Duración (Dias)
TFG	01/03/18	20/06/18	Alto	112
GEP	01/03/18	29/03/18	Bajo	29
Definición del alcance y contextualización	01/03/18	04/03/18	Bajo	4
Planificación temporal	05/03/18	08/03/18	Bajo	4
Gestión económica y sostenibilidad	09/03/18	13/03/18	Bajo	5
Presentación preliminar	14/03/18	24/03/18	Bajo	11
Condiciones de especialidad	25/03/18	29/03/18	Bajo	5
Documento final y presentación oral	25/03/18	29/03/18	Bajo	5
Familiarización con el entorno	07/03/18	17/03/18	Bajo	11
Instalación entorno de desarrollo	18/03/18	23/03/18	Bajo	6
Diseño e implementación	24/03/18	19/05/18	Bajo	57
Reconocimiento del hardware	24/03/18	31/03/18	Medio	8
Diseño	24/03/18	27/03/18	Medio	4
Implementación	28/03/18	31/03/18	Medio	4
Integración de la comunicación con el hardware	01/04/18	08/04/18	Bajo	8
Diseño	01/04/18	04/04/18	Bajo	4
Implementación	05/04/18	08/04/18	Bajo	4
Generalización de funciones	09/04/18	16/04/18	Bajo	8
Implementación	09/04/18	12/04/18	Bajo	4
Diseño	13/04/18	16/04/18	Bajo	4
Gestión de memoria	17/04/18	02/05/18	Alto	16
Diseño	17/04/18	22/04/18	Alto	6
Implementación	23/04/18	02/05/18	Alto	10
Scheduler	03/05/18	19/05/18	Alto	17
Diseño	03/05/18	08/05/18	Alto	6
Implementación	09/05/18	14/05/18	Alto	6
Testeo de funcionalidad	20/05/18	15/06/18	Medio	27
Memoria del trabajo	26/03/18	20/06/18	Bajo	87

Figura 5: Tabla de gestión de tiempo

Como hemos podido observar, la duración del proyecto total son 112 Días, en las cuales idílicamente tendríamos que trabajar durante 375h para completar los 18 créditos de la asignatura, esto nos da que idílicamente deberíamos trabajar 3,3h al día durante este tiempo.

No obstante, para ser prudentes y suponiendo retrasos debido a las dificultades añadidas que puedan surgir, y a que el proyecto puede durar más que las horas previstas por la asignatura, aumentaremos en un 20% el tiempo dedicado.

Cómo trabajaremos durante 112 días y tendremos que cumplir 450h, cada día deberemos trabajar un total de **4h,** o el equivalente a un trabajo a media jornada.

En la siguiente figura, se puede observar un *gannt* de la distribución del tiempo. El orden de las tareas asignadas es el mismo que en la tabla anterior.

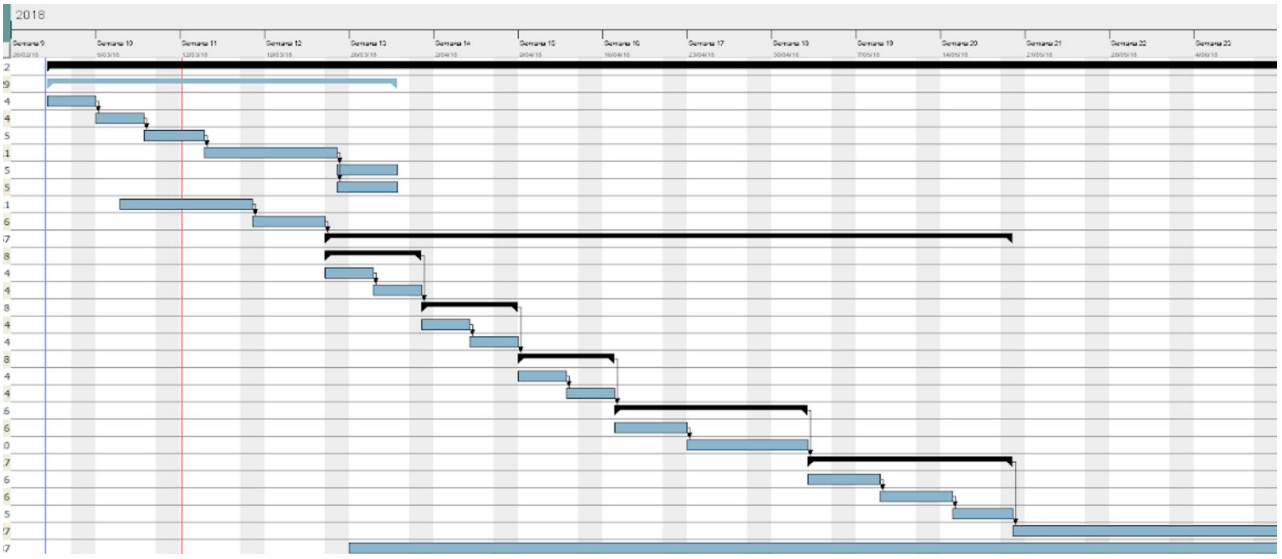

Figura 6: Diagrama de *Gannt*

4.4 Riesgos y alternativas

En este proyecto, cualquier contratiempo puede complicar la realización de los diferentes apartados ya que se encuentran relacionados. Uno de los problemas más inherentes del desarrollo del mismo, es la dificultad de probar la funcionalidad final sin construir primero todas las piezas del puzzle.

Por ello, se dedicará tiempo a la creación de juegos de pruebas para las diferentes funcionalidades que se añadan al programa, de forma que sea más fácil encontrar posibles errores que puedan surgir.

También, uno de los cuellos de botella que presenta nuestro proyecto, es la dependencia directa que hay entre el desarrollo de la implementación para GPU y para FPGA, pues los patrones de diseños, al tener que ser uniformes, dependen los unos de los otros.

Una alternativa a la dependencia con la implementación de GPU es la posibilidad ya hablada con el director del proyecto de hacer una versión funcional utilizando mi propio diseño, sin depender del diseño de GPU, haciendo que, aunque en un futuro haya que rehacer la distribución, la funcionalidad ya esté implementada y funcionando.

5 Recursos

Para llevar a cabo este proyecto se necesitan una serie de recursos, tanto humanos como *hardware* y *software*, en los siguientes apartados los desglosamos.

Nota que no dividimos los recursos por tareas, porque en todas las tareas serán útiles en mayor o menor medida cada uno de los recursos referenciados.

5.1 Recursos humanos

Recurso	Justificación
Director y codirector	Seguimiento, supervisión y ayuda en el proyecto
Empleados BSC	Ayuda en el desarrollo e implementación
Desarrollador	Encargado del proyecto

Figura 7: Recursos humanos

5.2 Recursos hardware

Recurso	Justificación
Ordenador portátil	Necesario para la ejecución y programación del modelo, así como para grabar la presentación para la asignatura de GEP.
Placa de desarrollo con FPGA	Necesario para poder testear la implementación

Figura 8: Recursos hardware

5.3 Recursos software

Recurso	Justificación
Sistemas operativos	Se utilizará Windows 10 y Ubuntu
Control de versiones	Se utilizará el Gitlab interno del BSC
Debugger	Se utilizará GBD, entre otros.
Editores de texto	Se utilizará Visual Studio, Notepad++ y VIM
Modelo de programación OmpSs-2	Se utilizará Mercurium y la versión modificada de Nanos6.
Otros	Utilidades online, como google docs, google calendar, y potencialmente overleaf.

Figura 9: Recursos software

6 Gestión Económica

6.1 Recursos humanos

En la siguiente tabla se pueden ver reflejados los diferentes recursos humanos que serán necesarios para la realización del proyecto. la estimación de horas del Director y Co-Director ha sido suponiendo la realización de una reunión semanal de dos horas cada una.

Los costes en recursos humanos son orientativos y pueden diferir de la realidad, el desarrollador, al ser yo mismo, no tiene remuneración.

Recurso	**Dedicación(horas)**	**Precio/hora(€/h)**	**Coste total(€)**
Director	32	20	640
Co-director	32	20	640
Trabajadores BSC	16	10	160
Desarrollador	450	0	0
Total	530	-	1440

Figura 10: Tabla de costes en recursos humanos

6.2 Recursos de software

En la siguiente tabla se pueden ver reflejados los diferentes recursos de *software* a utilizar. Para nuestro proyecto, podremos observar que todo el *software* es utilizado, en caso que no sea gratuito, en su versión gratuita.

En caso de Windows 10 Home, el precio de la licencia se incluye en el precio del portátil, incluído como recurso hardware.

Como para este proyecto, el *software* no cuesta dinero, no hay amortización posible.

.

Recurso	**Precio (€)**	**Unidades**	**Vida útil (años)**	**Amortización (€/h)**
Ubuntu	0	1	3	-
Windows 10 Home	0	1	3	-
Nanos6	0	1	-	-
Mercurium	0	1	-	-
Visual Studio	0	1	-	-
Notepad++	0	1	-	-
Gitlab	0	1	-	-
Gantter	0	1	-	-
Google Docs	0	1	-	-
Total	0	-		

Figura 11: tabla de recursos software

6.3 Recursos hardware

Para la realización del proyecto utilizaremos un portátil Cube thinker i35 y un ordenador con una FPGA conectada.

En el caso del portátil, pertenece al desarrollador, que lo utiliza como ordenador de uso personal. Se ha calculado cuánto tiempo se utilizará el portátil para este proyecto y cuánta parte del coste del mismo repercute sobre el proyecto.

En el caso del ordenador con FPGA, es cedido por el BSC para la realización de los test de la implementación. Debido a que es un hardware cedido, no consideraremos el precio del mismo para este proyecto.

Recurso	Precio	Unidades	Vida útil	Amortización
Cube thinker i35	631€	1	4 años	82€
Xilinx FPGA board 702	-	1	-	-
Total	**631€**	**-**		**82€**

Figura 12: Tabla de recursos hardware

6.4 Costes indirectos

El proyecto se desarrollará en instalaciones de la UPC, ya sea el edificio Omega, sala de estudio o biblioteca, por ello, deberemos incluir el transporte hasta la universidad como coste indirecto y el gasto teórico orientativo de la utilización de recursos en la universidad, como por ejemplo, internet, agua y electricidad.

Coste	**Precio**	**Duración**	**Coste total**
TJove 3 zonas	66.4€/mes	4	265.6€
Gastos esp. trab.	45€/mes	4	180€
Total	100		445,6€

Figura 13: Tabla de costes indirectos

6.5 Control de gestión

En un proyecto de *software* una de los principales motivos de salida de presupuesto es la disociación entre el tiempo estimado para una tarea y el tiempo real que dura esa tarea. Debido a ello, es importante llevar un registro del tiempo real de duración de cada parte del proyecto, pues es necesario para poder justificar el coste real.

Otra forma de asegurar que los recursos empleados en las diferentes tareas no sea en vano, es las reuniones semanales con el director del TFG que guía en el proyecto y si hay alguna tarea que se está implementando o diseñando con fallas, puede ser corregida sin repercutir más de la cuenta con el plan de tiempo original. De igual manera, para asegurar que no queda trabajo pendiente y para organizar el espacio de trabajo, se utilizará una herramienta denominada **trello**[14], que nos permitirá definir tareas y, visualmente, agruparlas según si están por hacer, aparcadas, on-going o finalizadas.

Debido a que en el proyecto no se espera que haya diferencias en el *software* a usar, y si se utiliza *software* adicional se buscará que sea de libre uso, no debería haber diferencias en cuanto a presupuesto estimado y final. De igual manera, no se espera que se tenga que utilizar más *hardware* que el anteriormente descrito.

Para calcular las desviaciones de costes, se utilizará simplemente la fórmula:

$$Desviación = [coste\ estimado] - [coste\ real]$$

Teniendo en cuenta que para calcular el coste de una tarea, necesitamos calcular la Desviación en horas dedicadas y multiplicarlo por el precio hora que tenemos en apartados anteriores.

$$Desviación_{coste} = ([tiempo\ estimado] - [tiempo\ real]) * precio_{hora}$$

Tras ello, y, tal como hemos dicho en un principio, al tratarse de un proyecto de *software*, en el que el tiempo dedicado puede fluctuar mucho, y dificultades añadidas pueden hacer que se requieran más horas de los recursos humanos disponibles, se ha decidido una contingencia de un 15% para prevenir dentro del presupuesto los riesgos que pueda generar nuestra predicción.

[14] "Trello." https://trello.com/. Se consultó el 19 mar.. 2018.

6.6 Presupuesto

A continuación, se presenta un desglose del coste directo y un presupuesto para el proyecto, juntado con las tareas del gantt de la entrega anterior.

Tarea	Tiempo (Horas)	Amortización Portátil
TFG	**450**	**82€**
GEP	75	14€
Familiarización con el entorno	23	4€
Instalación entorno de desarrollo	13	2€
Diseño e implementación	119	23€
Reconocimiento del hardware	16	3€
Integración de la comunicación con el hardware	16	3€
Generalización de funciones	16	3€
Gestión de memoria	33	6€
Scheduler	35	6€
Testeo de funcionalidad	56	11€
Memoria del trabajo	38	7€

Figura 14: Desglose del coste directo.

Hoja de presupuesto				Total
Coste directo				**82€**
	recurso	**Amortización**	**Tiempo (horas)**	
	Cube Thinker i35	**0.17**	**450**	**82€**
Coste indirecto				**445.6€**
Coste RRHH				**1440€**
Total sin impuestos				**1967.6€**
Contingencia 15% sobre el Total sin impuestos				**295.14€**
Total sin IVA				**2262.74€**
Total con IVA 21%				**2737.92€**

Figura 15: Presupuesto para el proyecto

7 Sostenibilidad

7.1 Matriz de sostenibilidad

	PPP	Vida Útil	Riesgos
Ambiental	Consumo del diseño	Huella ecológica	Ambientales
	9:10	17:20	0:0
Económico	Factura	Plan de Viabilidad	Económicos
	8:10	18:20	0:0
Social	Impacto personal	Impacto social	Riesgo social
	8:10	10:20	0:0
Rango sostenibilidad	25:30	45:60	0:0
	70:90		

Figura 16: Matriz de sostenibilidad

7.2 Dimensión Económica

Tras la valoración de costes tratada en el apartado de gestión económica, podemos observar que el *software* utilizado para el desarrollo del proyecto no es relevante.

El *hardware* a utilizar para el desarrollo, no únicamente es totalmente reaprovechable para cualquier otro proyecto u ocio, sino que ya se estaba en posesión de la misma antes de plantearse este proyecto.

Los recursos humanos en este caso es donde el coste del proyecto tiene su mayor peso, pues al ser un proyecto de desarrollo de *software*, en el que hay varios actores implicados (Director, Co-Director y trabajadores del BSC) los cuales, en sus horas de trabajo, dedican tiempo para la realización de este proyecto.

Respecto al impacto económico del proyecto, la realización del mismo implica, en primera instancia, un ahorro económico, pues al ser un TFG las horas del desarrollador no son remuneradas. Mientras que si la implementación la hiciese un trabajador del centro, tendría un sueldo.

Otro impacto económico se puede apreciar a que si hay un mejor modelo de programación, que permita a los programadores que tengan que tratar con él ser más eficientes, y por lo tanto, tardar menos tiempo en la programación de algoritmos que utilicen el modelo, la suma de horas ahorradas de sueldo se consideran como beneficio.

7.3 Dimensión Social

El modelo de programación OmpSs-2 es un modelo de programación cuyo código fuente puede ser encontrado en github, siguiendo la página web del bsc.

Debido a ello, usuarios y empresas que quieran utilizar este modelo de programación para acelerar su trabajo, podrán hacerlo, mejorando su experiencia tratando con sistemas heterogéneos.

El proyecto, no obstante, a nivel social seguramente tenga un impacto nulo, pues está destinado a ser utilizado internamente por el BSC y por aquellos con conocimientos técnicos.

La necesidad de este proyecto nace de facilitar el uso de FPGAs utilizando modelos de programación, de momento, las FPGAs son utilizadas en ámbitos donde el tiempo real es muy necesario, como por ejemplo, impresoras, pero últimamente se está intentando incorporar su uso para hacer cálculos, que, aunque de momento cuesta de justificar su uso sobre el de una GPU, la versatilidad y la mejora continua de las FPGAs hacen que en un futuro pueda ser una potente herramienta de cálculo.

7.4 Dimensión Ambiental

El impacto ambiental que supondrá el proyecto será casi nulo, teniendo como impacto la utilización de un ordenador, la luz del espacio de trabajo y el viaje en autobús.

Los recursos *hardware* son totalmente reutilizables para cualquier otro proyecto.

Nuestro proyecto impacta en el medio ambiente de una manera sutil, pues como estaremos facilitando el uso de ciertos dispositivos gracias al modelo de programación, el tiempo ahorrado de programación por quienes lo usen, será tiempo que un computador no tiene por qué estar encendido, no obstante, no se espera un impacto ambiental real.

8 Diseño y implementación

Para poder entender los cambios en el diseño del *runtime*, en esta sección se explicará la funcionalidad de las partes implicada, posteriormente, se analizarán los cambios realizados para la implementación.

Se han modificado cuatro piezas de nanos, **Hardware, Scheduler, Executors** y la **API.**

8.1 Estado del runtime

Antes de explicar los cambios realizados al *runtime* Nanos6, parece conveniente explicar las diferentes estructuras y la forma en que se conectan entre sí. De esta forma, se pretende dar una visión global de las partes implicadas en la ejecución de una tarea, para poder entender luego los cambios realizados.

8.1.1 Hardware

El *runtime* necesita conocer las características de la plataforma en dónde se está ejecutando para poder tomar decisiones a la hora de ejecutar las peticiones que recibe.

Para ello, existe un concepto llamado ***HardwarePlace*****,** una estructura que identificará a un tipo de hardware, sea de cálculo llamado *ComputePlace* o de memoria llamado *MemoryPlace*.

El ***ComputePlace*** consiste en cualquier *hardware* de procesamiento capaz de ejecutar una tarea, en la versión del *runtime* utilizada como base, únicamente existe el *hardware* **CPU**, pero una **GPU** o **FPGA** también entrarían dentro de este concepto.

El ***MemoryPlace*** consiste en Cualquier *hardware* en el que se necesita almacenar los datos para poder ejecutar una tarea. Un ejemplo de memory place son los NUMA nodes de la máquina, o la RAM privada de algunos dispositivos como GPU o FPGA.

A continuación se puede apreciar una figura en la que se muestra las subclases derivadas de *HardwarePlace* y su estructuración dentro del *runtime*.

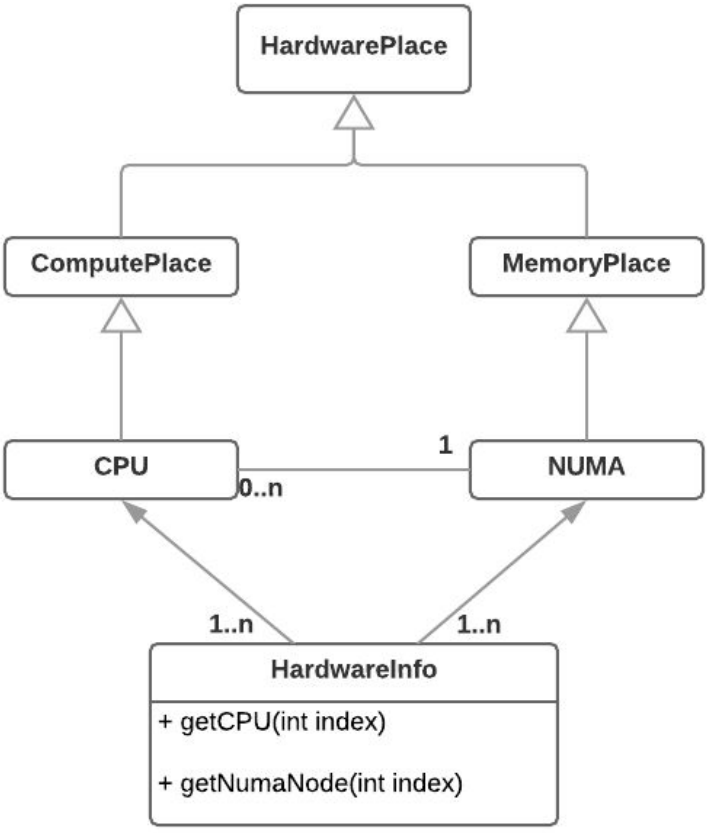

Figura 17: Diagrama UML de las estructuras de los tipos de hardware soportados.

8.1.2 Executors

La ejecución de los tareas son llevadas a cabo por los ***Worker Threads***. Por defecto, se genera un thread por CPU disponible, al que se le adjudica un Worker Thread. Son los encargados de ejecutar cada una de las tareas que se han enviado al *runtime*, por lo que siguen el siguiente esquema en bucle infinito:

1.- Obtener tarea *ready* del Scheduler

2.- Ejecutar el código de usuario de la tarea

3.- Liberar las dependencias de datos

Este comportamiento puede ser apreciado en la siguiente figura:

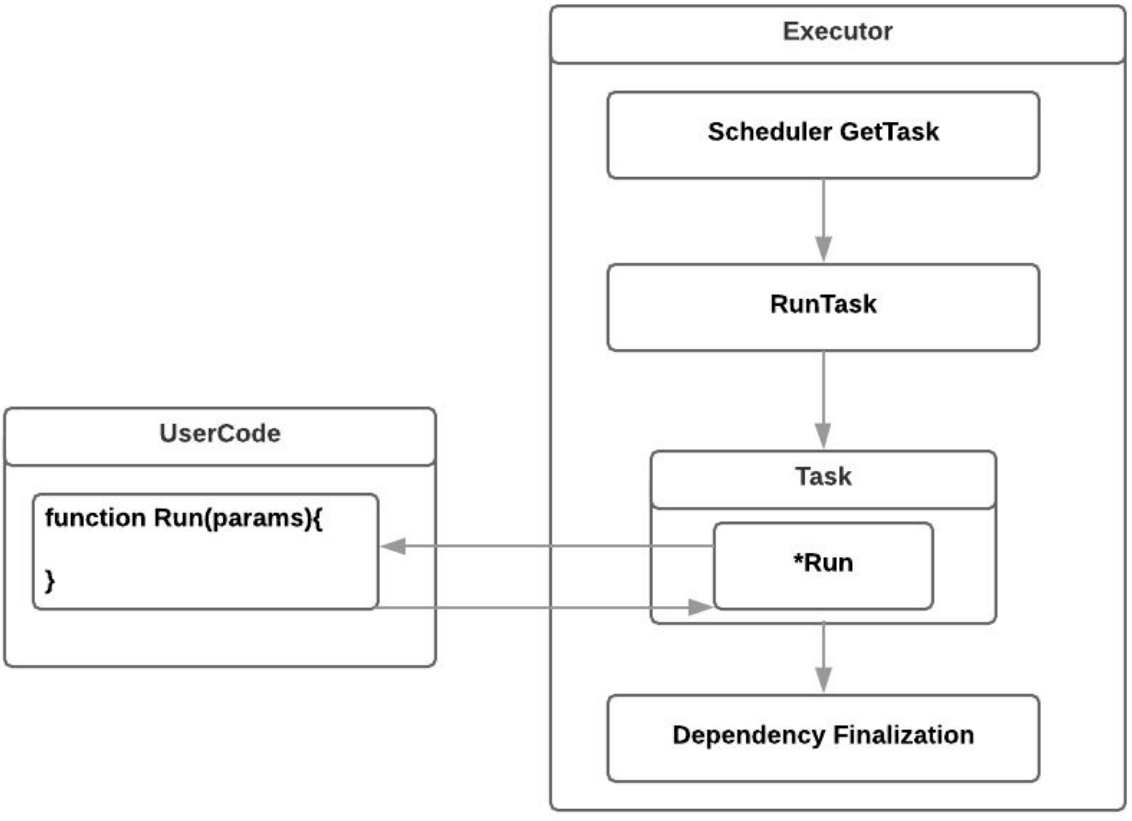

Figura 18: Esquema de funcionamiento del executor.

Un *thread*, en caso de no encontrar ninguna tarea para ser ejecutada se van a un estado *Idle* en una cola del *scheduler*.

Este comportamiento tiene como fin el no inundar con peticiones de tareas que no pueden ser atendidas la CPU. Los *threads* en *Idle* se volverán a activar por el scheduler, mediante un ***ThreadManager*****,** encargado de controlar todos los threads del sistema.

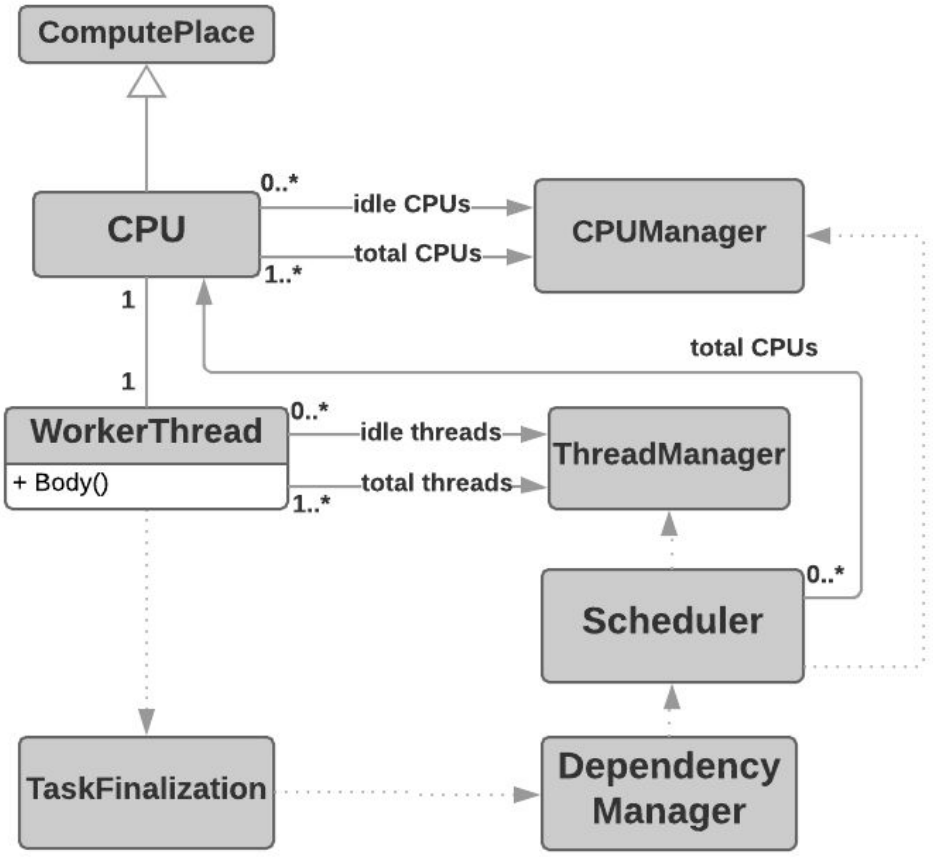

Figura 19: Diagrama que muestra el mecanismo de petición de threads, finalización y dependencias.

Polling Services

Los *polling services* tienen la tarea de sincronizar métodos asíncronos, cuyo cometido es no bloquear la ejecución del *thread* actual, a la espera que cuando se necesite el dato el método ya haya terminado.

Los *polling services* entran en acción cuando un *thread* está en *idle* y justo antes de ejecutar una tarea.

```
typedef int (*nanos_polling_service_t)(void *service_data);

void nanos_register_polling_service(char const *service_name,
             nanos_polling_service_t service_function, void *service_data);

void nanos_unregister_polling_service(char const *service_name,
             nanos_polling_service_t service_function, void *service_data);
```

Figura 20: API de los nanos polling services

La filosofía detrás de los polling services es la de ejecutar una función cada vez que un *thread* no tiene trabajo o va a coger trabajo, de forma que se intenta que entorpezca lo menos posible la ejecución real de una aplicación, ya que no puede interrumpir.

para registrar una nueva función, hay que llamar a **nanos_register_polling_service,** tras esta llamada, nuestra función empezará a ser ejecutada por los hilos.

Si queremos desactivar servicio, por ejemplo, mientras no hay ninguna tarea ready, podemos llamar a **nanos_unregister_polling_service** para evitar consumir tiempo de CPU.

8.1.3 Scheduling

El *scheduler* es una estructura que nos permite añadir tareas que están listas para ejecutarse a una cola, de la cual saldrán según la lógica de scheduling, al ser requerida una tarea para ser ejecutada.

Una tarea llega al scheduler cuando el sistema de dependencias decide que una tarea está lista para ser ejecutada, y, mediante la API del *scheduler*, pone la tarea en la cola de *ready*.

Una vez en *ready*, esperará que un *Worker Thread* utilice su API para pedir una tarea, y retornará una tarea adecuada a la petición, respetando la política de *scheduling* elegida.

En el caso que no haya *hardware* suficiente disponible para los requisitos de la tarea, o no hay una tarea que ejecutar, marcará el thread que ha pedido la tarea como idle.
Por lo tanto, el Scheduler gestionará los diferentes hilos de peticiones, activando en caso de que se añada una nueva tarea hilos en estado idle para poderles suministrar la tarea.

A continuación se puede apreciar una figura en la que se muestra las clases que implementan *SchedulerInterface*.

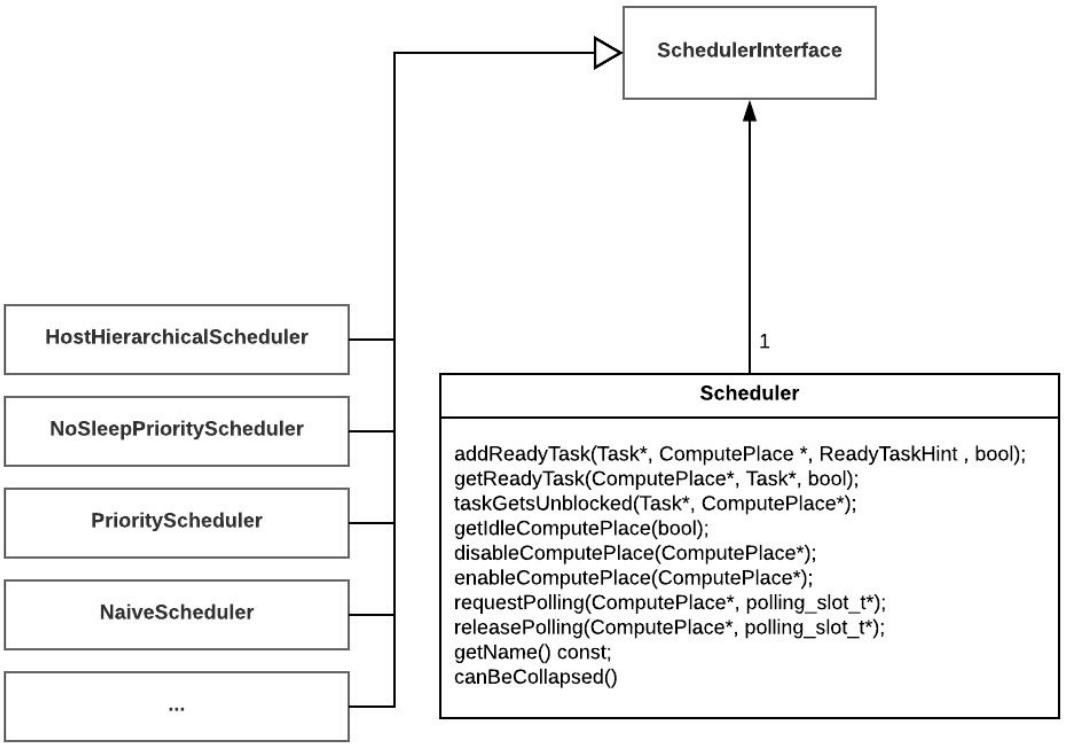

Figura 21: Clases y subclases de SchedulerInterface

La clase *Scheduler* contiene únicamente un objeto instancia de ***SchedulerInterface***, que contendrá la lógica de *scheduling* según el *Scheduler* que hayamos decidido crear. Para la implementación de FPGA utilizaremos una instancia de *HostHierarchicalScheduler*.

Nanos6 hace uso de diversas **variables de entorno** para controlar el funcionamiento del *runtime* en tiempo de ejecución. Relacionado con el *Scheduler* tenemos dos variables de entorno que pueden modificar el funcionamiento del mismo.

Variable de entorno	Descripción
NANOS6_SCHEDULER	Determina el tipo de *Scheduler*
NANOS6_DEVICE_SCHEDULER	Determina las hojas del *Scheduler*.

Figura 22: Variables de entorno relacionadas al tipo de scheduler

Las llamadas más importantes a la **API** del scheduler son **addReadyTask** y **getReadyTask**.

addReadyTask es llamada por el sistema de dependencias para añadir una tarea cuyas dependencias hayan sido resueltas. Devuelve el *ComputePlace* sobre el que se ejecutará la tarea.

getReadyTask es llamada por un *thread* cuando requiere de una tarea para ejecutar. Recibe como parámetro un *ComputePlace* para saber qué tipo de tarea debe devolver.

8.1.4 API Nanos6

La API de Nanos6 es la forma que tiene de comunicarse el compilador Mercurium, o un programador que esté usando Nanos6 como librería con el *runtime*.

La API expone las diferentes estructuras que son necesarias para hablar con el *runtime*, así como las llamadas que se pueden hacer al mismo.

Las llamadas que pueden ser consideradas más relevantes y necesarias para nuestro proyecto son **nanos_create_task, nanos_submit_task** y **nanos_taskwait.**

API CALL	DESCRIPCIÓN
nanos_create_task	Genera una tarea en el *runtime* y devuelve un handler a la misma.
nanos_submit_task	Envía una tarea generada con nanos_create_task, cuando se hace esta llamada, la tarea entrará al sistema de dependencias y pasará a estar ***ready*** cuando no haya un bloqueo por dependencias de datos.
nanos_taskwait	Simula una tarea con dependencias a todas las tareas que están en el *runtime*, cuando el *runtime* dice que las dependencias para ella están ready, la función devuelve el control al usuario. Por lo tanto, llamar a esta función hace que finalicen todas las tareas con dependencias.

Figura 23: API de Nanos6 referente a la creación y espera de tareas

Por otra lado, la estructura más relevante para nuestra modificación es el **nanos6_task_implementation_info_t**.

ATRIBUTO	**DESCRIPCIÓN**
device_type_id	El tipo de dispositivo en el que ejecutará la tarea nanos6_host_device = 0 nanos6_cuda_device = 1 nanos6_opencl_device = 2 nanos6_fpga_device = 3
run	Puntero a la función de código de usuario que se ejecutará por el *runtime*.
get_constraints	Función para recibir las restricciones de la tarea, por ejemplo, de memoria, coste, energía...
task_label	Descripción de la tarea
declaration_source	Línea de código en la que la tarea se ha instanciado

Figura 24: Parámetros de la implementación del task para la API de Nanos6

8.1.5 Mapa de ejecución de una tarea

En el siguiente diagrama, se puede apreciar la vida de una tarea SMP desde que el usuario la crea hasta que es ejecutada y finalizada por el *runtime*.

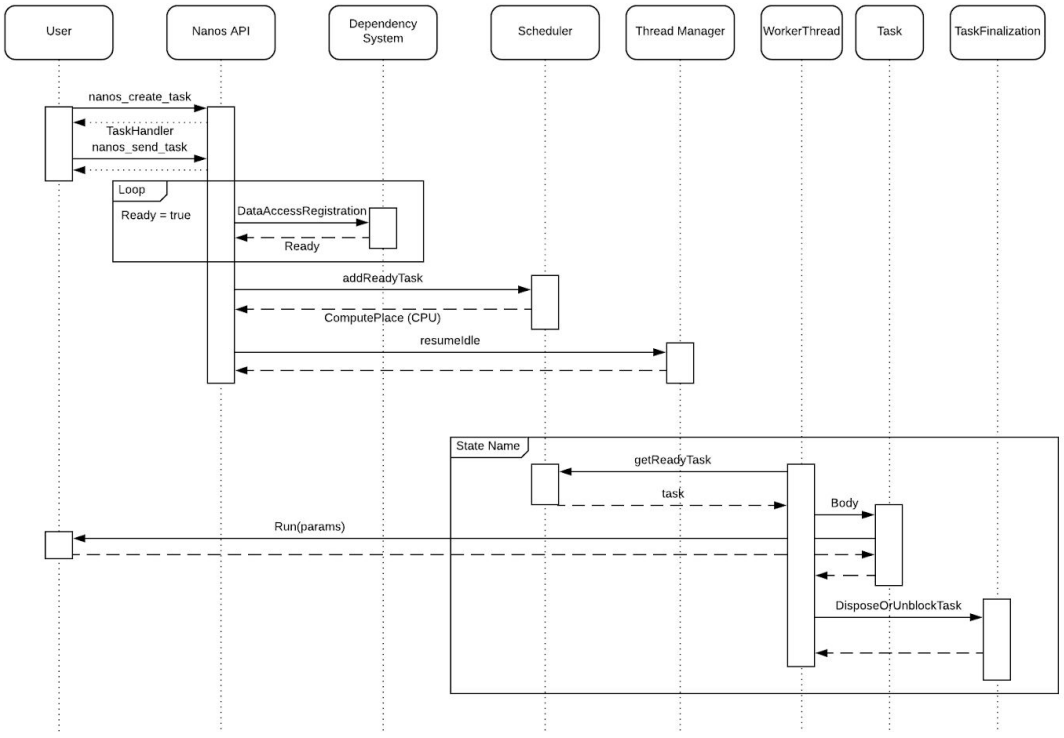

Figura 25: Esquema de ejecución de una tarea SMP hasta su finalización

8.2 Nanos6 Cambios para soportar FPGA

Con tal de lograr la ejecución de código en la FPGA, se han realizado una serie de cambios que vamos a tratar en este apartado.

En algunos de los diagramas que encontraremos en este apartado, se puede observar como hay partes del esquema con un color verde, esto está hecho para mostrar estructuras o funcionalidades añadidas para implementar el soporte.

8.2.1 Comunicación con FPGA

Como hemos introducido en la contextualización, una FPGA es un dispositivo *hardware* programable.

Para poder comunicarnos con este hardware, utilizaremos dos librerías que se encargarán de la comunicación con el controlador del dispositivo, para así abstraer en una interfaz común para todas las FPGA.

Además de ello, hay que tener en cuenta el modelo de ejecución. Mientras que para ejecuciones en CPU conocemos que hemos finalizado una tarea tras acabar de ejecutar el código de usuario, es decir, ejecutamos de manera síncrona, para FPGA, la comunicación con el *hardware* la haremos de manera asíncrona.

Esto es debido a que no precisamos de hacer uso de la CPU, la dejaremos libre mientras que la ejecución se hace en la FPGA. Por ello, deberemos comprobar la finalización de la ejecución de la tarea y liberar sus dependencias cuando haya acabado.

Para poder comunicarnos con el hardware, las librerías que utilizaremos son **LibXtasks y LibXDMA**, las cuales detallaremos su funcionamiento en el próximo apartado.

8.2.2 LibXDMA

LibXDMA es un *driver* y librería que permite hacer transferencias **xdma** (*Xilinx Direct Memory Access*) entre dispositivos.

DMA es una forma de transferir datos entre dispositivos sin que la CPU tenga que intervenir, de manera que liberamos la carga de trabajo de la CPU, a consecuencia, los datos que se tienen que enviar no deben estar cacheados, o la transferencia tendrá valores no consistentes.

Para ello, dispone de una Interfaz de la que definiremos las llamadas más relevantes.

API	**DESCRIPCIÓN**
xdmaAllocateKernelBuffer	Reserva memoria ***pinned***, esta memoria es no cacheable, por lo que cualquier intento de leer/escribir en ella siempre será fallo de cache. Devuelve un *handler* a esta memoria.
xdmaFreeKernelBuffer	Libera memoria ***pinned*** del buffer referenciado en el handler pasado como parámetro.
xdmaGetDMAAddress	Te devuelve la dirección física de un buffer ***pinned*** al pasarle como parámetro el *handler* de la función allocate.
xdmaSubmitBuffer	Envia un *buffer* reservado con el allocate al dispositivo que le pasemos como parámetro.
xdmaWaitTransfer	Para una transferencia, se bloquea hasta que esta acaba.

Figura 26: API de LibXDMA

8.2.3 LibXtasks

LibXtasks es una librería que tiene como objetivo abstraer la comunicación con la FPGA. Utiliza LibXDMA internamente para la comunicación. Además, necesita un fichero de configuración con el device tree de los diferentes aceleradores FPGA del sistema.

A continuación, se encuentra una tabla con algunas de las llamadas a la API más importantes.

API	**DESCRIPCIÓN**
xtasksGetNumAccs	Devuelve el número de aceleradores FPGA del sistema
xtasksGetAccs	Devuelve un *handler* para poder acceder a cada acelerador del sistema.
xtasksGetAccInfo	Para un *handler*, devuelve información sobre el tipo de acelerador del que se trata y su frecuencia.
xtasksCreateTask	Genera una tarea, contiene una ID como atributo, devuelve un *handler* para la tarea.
xtasksDeleteTask	Libera estructuras creadas por xtaskCreateTask para un *handler*.
xtasksAddArg	Añade a una tarea argumentos. Es importante notar que los argumentos tienen que ser direcciones de **memoria *pinned*** para que funcione.
xtasksSubmitTask	Envia una tarea que será ejecutada por el acelerador seleccionado
xtasksTryGetFinishedTask	Intenta recibir la información de cualquier tarea que haya acabado. Devuelve el handler y la ID de la tarea acabada.

Figura 27: API de LibXtasks

8.2.4 Modelo de ejecución en FPGA

Una de las diferencias clave de la ejecución en un dispositivo FPGA respecto a una ejecución en una CPU es la notificación de acabado.

La lógica de control del *runtime* se encuentra en la CPU, que es compartida por las tareas SMP. En el caso de una tarea SMP, la finalización de una tarea es detectada al volver de la llamada a la función de usuario, la cual se encarga de la ejecución.

Para el caso FPGA, debemos evitar monopolizar la CPU durante el tiempo que estamos ejecutando nuestro programa en el acelerador, pues la CPU podría estar haciendo trabajo útil.

En la siguiente figura se puede ver un diagrama de un usuario intentando ejecutar una función en una FPGA. Para la simplicidad del diagrama, se han obviado las llamadas que hace Xtasks a Xdma y se ha supuesto un entorno ya inicializado.

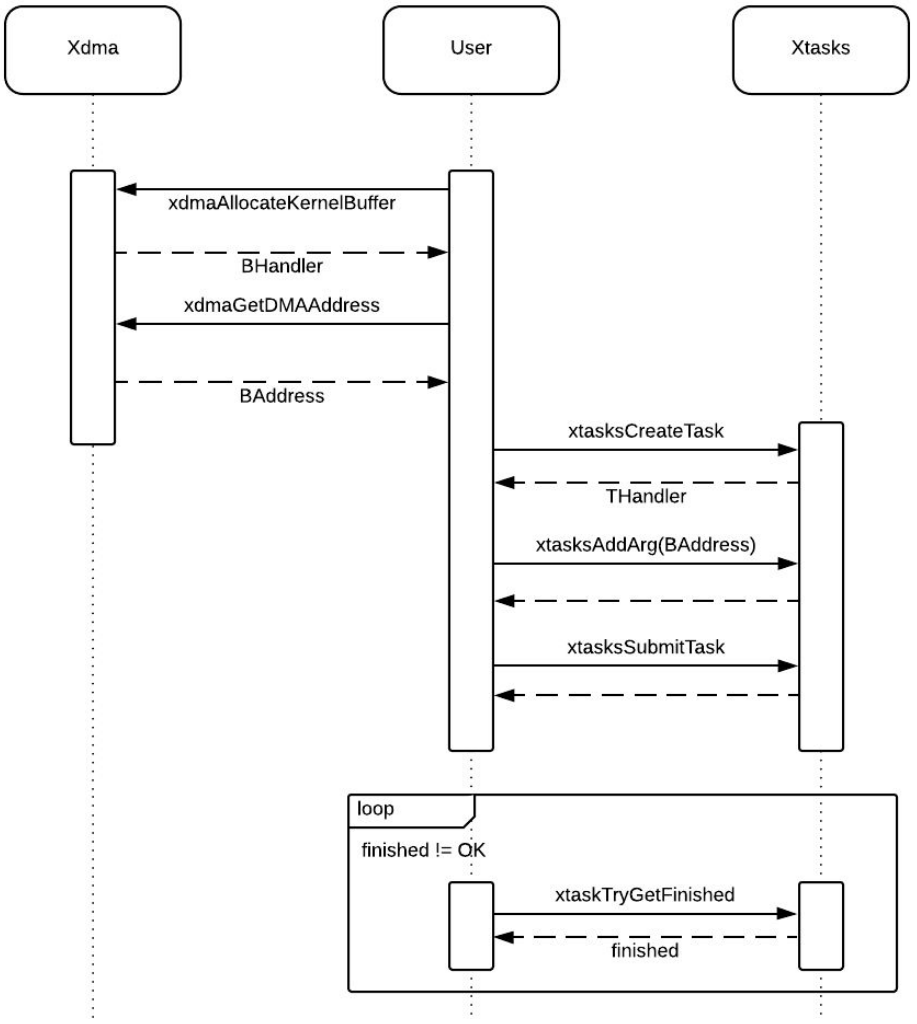

Figura 28: Diagrama de ejecución de una tarea utilizando Xdma y xtasks

Una descripción de los pasos que haría un usuario para poder ejecutar una aplicación FPGA sin usar el *runtime* serían:

1.- Pedir memoria de **kernel *pinned*** mediante la llamada xdmaAllocateKernelBuffer (Fig. 6).
2.- Pedir la dirección de memoria *hardware*.
3.- Inicializar el buffer reservado a los valores que queramos enviar a la FPGA.
4.- Crear una nueva tarea.
5.- Añadir como parámetro la dirección hardware.
6.- Hacer *submit* de la tarea.
7.- Esperar a que finalice la ejecución.

Para poder llegar a conseguir este objetivo en nuestro *runtime*, Nanos6 tiene que contemplar el modelo de memoria y ejecución para FPGA.

- La FPGA comparte memoria con el procesador, pero esta memoria es *pinned* y no cacheable.
- En el estado actual del soporte de Nanos6 a FPGA, el usuario es el encargado de reservar y liberar memoria para FPGA, en ese caso, el *runtime* tiene que actualizar la lista de direcciones.
- El *runtime* mantiene una lista de todas las direcciones físicas de la memoria utilizada para la FPGA junto con la dirección de memoria en el proceso de usuario, necesaria para realizar las copias.

Debido a la compartición de memoria con el procesador, no debemos modificar el sistema de dependencias actual para adaptarlo a nuestro caso.

Las direcciones administradas al sistema de dependencias son válidas para nuestro proceso, y cuando una tarea finaliza, los datos ya están actualizados en memoria.

Para nuestro modelo de ejecución, vamos a suponer únicamente el caso de FPGA sin memoria privada. Es decir, Para cualquier ejecución, al finalizar la tarea vamos a disponer de los resultados en memoria del proceso.

Para ejecutar las tareas FPGA utilizaremos una herramienta del *runtime* llamada *polling services.*

Los *polling services* es un mecanismo que ofrece el *runtime* en el que podemos elegir una función que queremos que sea ejecutada por los *worker threads*.

Abusaremos de este sistema para la espera de la finalización de tareas enviadas a ejecutar a una FPGA.

8.2.5 Cambios en Hardware

FPGAManager

Con tal de poder controlar las estructuras necesarias para la implementación, se ha creado un *Manager*.

Este *manager* se encargará de la inicialización de los componentes, contendrá el directorio de traducción de memoria y el mecanismo de detección de finalización de ejecución. También contendrá el polling service encargado del control de ejecución del programa.

Cuando es inicializado, se identifican todos los aceleradores del sistema, se crean todas las estructuras necesarias para su control y se encarga de inicializar las librerías LibXtasks y libXDMA.

Por otro lado, con tal de dar soporte a la API, se encarga de la implementación del Malloc y Free para FPGA, además de llevar un contador de programas en ejecución para dar soporte a la llamada nanos_FPGAtaskwait.

FPGAKernel

Debido a que podemos tener más de un acelerador que disponga de la misma función, se ha añadido una clase llamada **FPGAKernel**.

FPGAKernel dispone de todos los *ComputePlaces* que comparten funcionalidad, así como el tipo de acelerador con el que tendrá que coincidir la tarea para que sea ejecutado por sus *ComputePlaces*.

FPGAComputePlace

Se encarga de la ejecución de tareas y de la liberación de una tarea finalizada.
Cada *FPGAComputePlace* pertenece a un Kernel determinado.

A continuación, se muestra un diagrama con las estructuras añadidas a la configuración inicial.

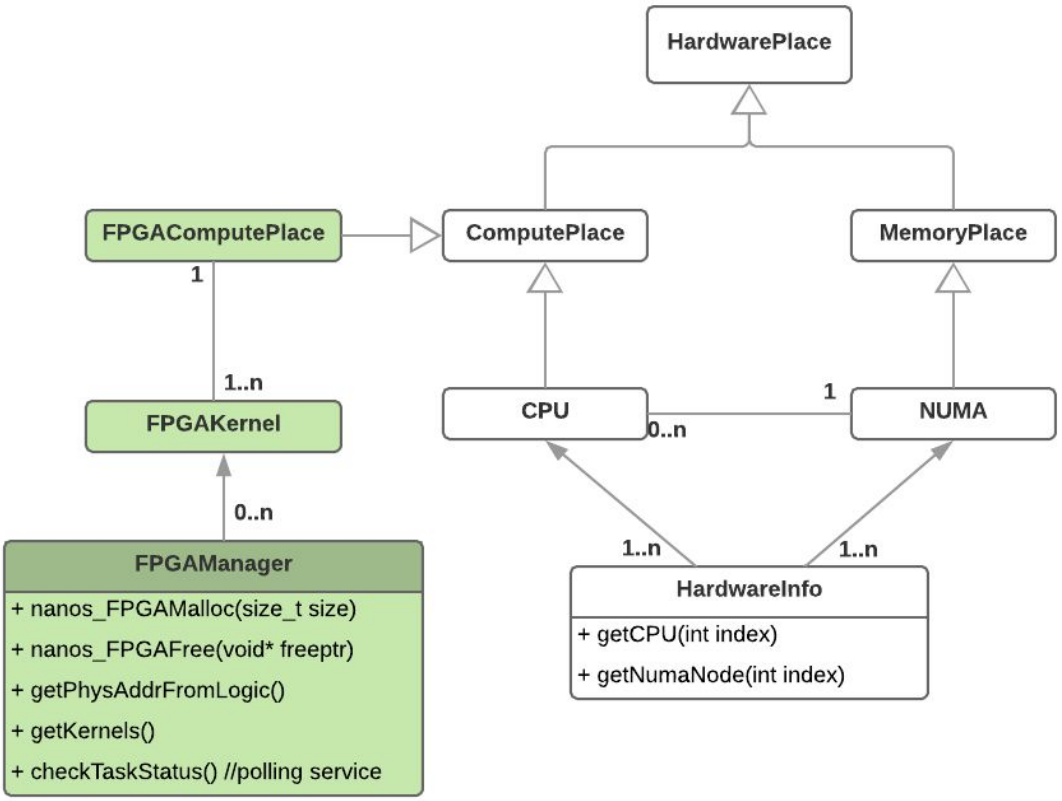

Figura 29: Esquema de Hardware con las clases añadidas en verde

8.2.6 Cambios en Executors

FPGAHelper

Con tal de suministrar tareas a los aceleradores FPGA, necesitamos que haya un servicio encargado de verificar si hay tareas que precisan ser servidas. Para ello, se ha diseñado el FPGAHelper, un servicio que se encargará de pedir tareas disponibles para ser ejecutadas al scheduler.

8.2.7 Cambios en la API

Para dar soporte a la implementación, se han creado tres nuevas funciones en la API de Nanos6. Estos cambios han sido mínimos y únicamente referentes a la gestión de la memoria y procesos FPGA en ejecución..

NUEVA FUNCIÓN	**DESCRIPCIÓN**
nanos_FPGAMalloc	Se comporta como una función malloc normal de cara al usuario, devolviendo una dirección en espacio de proceso en la que el usuario puede escribir. Internamente, llama a Xdma para pedir memoria *pinned*, y genera un directorio con la dirección física del buffer y el handler necesario para la librería.
nanos_FPGAFree	Se comporta como una función free normal de cara al usuario. Internamente, llama a Xdma para liberar memoria, además libera del directorio la memoria pedida.
nanos_FPGAtaskwait	Barrera que espera a que todas las tareas de FPGA hayan acabado.

Figura 30: Nuevas funciones en la API

8.2.8 Cambios en Scheduler

Tal y como hemos indagado, podemos entender el *Scheduler* como una estructura de datos en la cual, añadimos tareas que deben ser ejecutadas. Cuando hay *hardware* disponible para ejecutarlas, son obtenidas según la política de *scheduling*.

Para añadir la nueva estructura capaz de almacenar tareas de FPGA, se ha añadido un nuevo scheduler al *HostHierarchicalScheduler*, que hasta ahora únicamente contaba con tareas SMP.

Se ha creado en FPGAScheduler la lógica que separa las tareas según su tipo de acelerador y se ha hecho una implementación naive de un scheduler con una cola simple, la cual será la cola de ready para un tipo de acelerador FPGA concreto.

A continuación se puede ver un esquema de la implementación de la nueva funcionalidad.

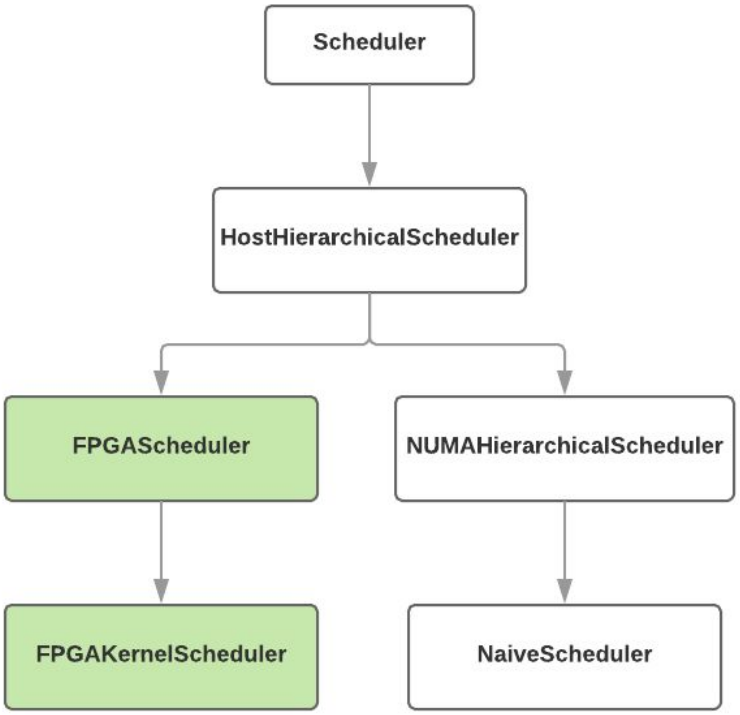

Figura 31: Esquema del Scheduler con las nuevas clases añadidas

En este caso, una tarea FPGA llegaría al objeto estático Scheduler, que llamaría a *HosHierarchicalScheduler*. este, al detectar que se trata de una tarea FPGA lo enviaría al *FPGAScheduler*, y éste lo encolaría en un *FPGAKernelScheduler* dependiendo el tipo del acelerador que se tratase.

8.2.9 Diagrama de ejecución FPGA en Nanos6

En los siguientes diagramas, se van a separar las tres funcionalidades básicas de la ejecución en el *runtime*.

La siguiente figura, detalla la instanciación de tareas para su posterior ejecución en FPGA. La diferencia respecto a la ejecución SMP es que el usuario tiene que pedir la memoria con anterioridad utilizando la nueva llamada a la API.

Al tratarse de una tarea FPGA, llegará al Scheduler correcto para su tipo de dispositivo.

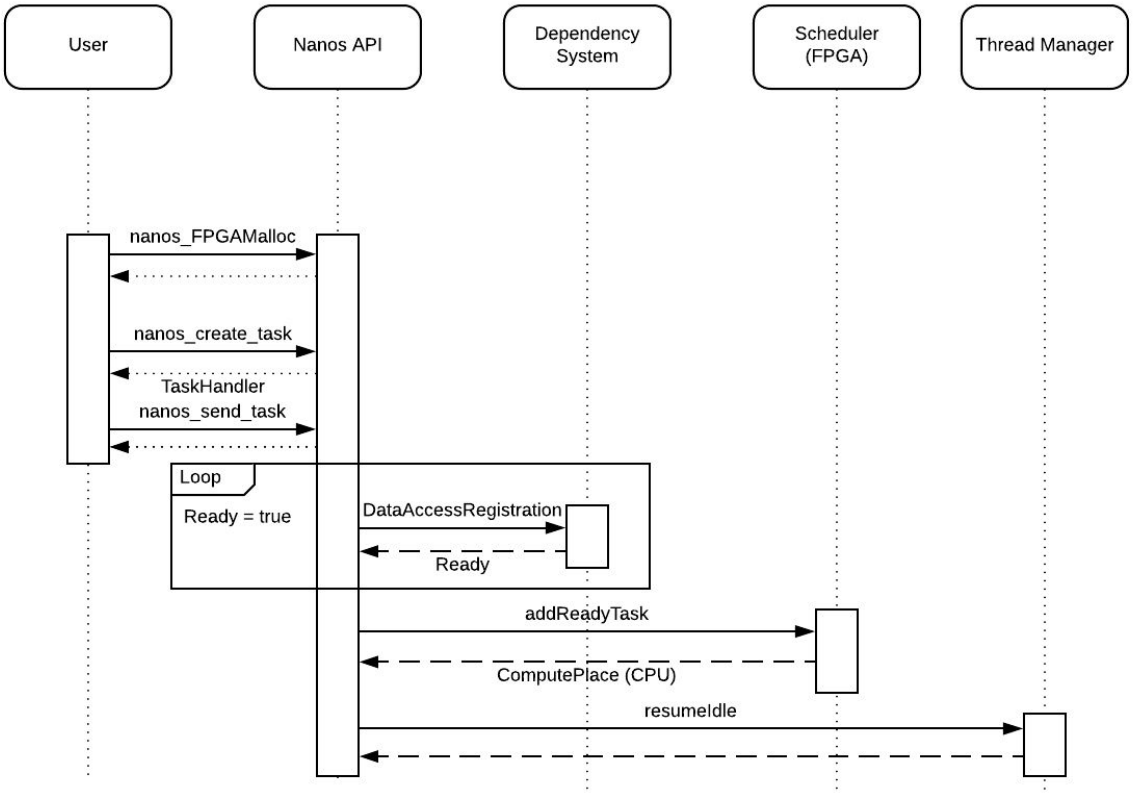

Figura 32: Esquema parcial que muestra el registro de una nueva tarea al *runtime*

En la siguiente figura, podemos ver en acción el *Polling Service* que se encarga de mandar a ejecutar tareas a una FPGA.

Para cada acelerador disponible, el polling service estará pidiendo tareas a ejecutar, preparando los parámetros y delegando en Xtasks su ejecución.

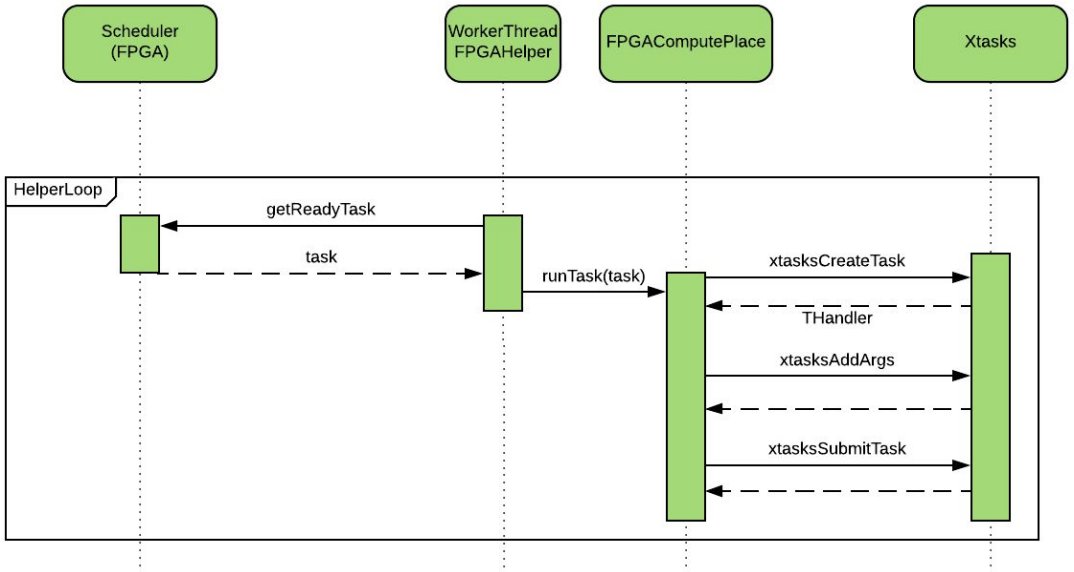

Figura 33: Esquema de la ejecución de una tarea en FPGA

Y en esta última, podemos apreciar el *Polling service* encargado de la detección del acabado de tareas de tareas y su finalización dentro del *runtime*.

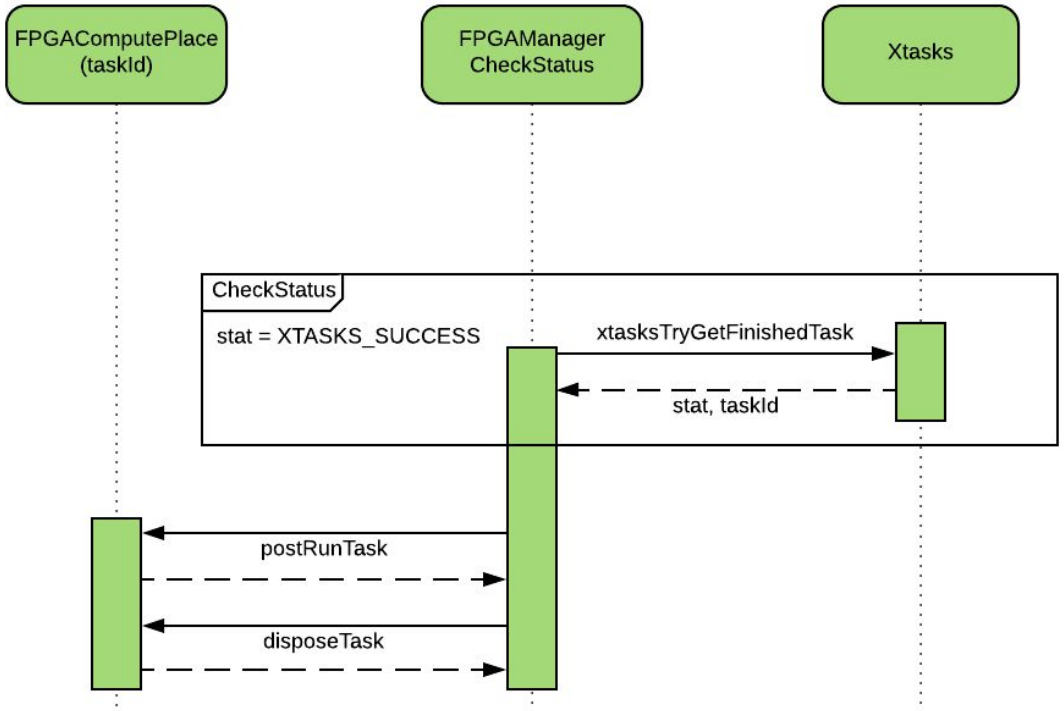

Figura 34: Esquema de la finalización de una ejecución de una tarea en FPGA

8.2.10 Cambios en el código de usuario

Para poder ofrecer una generación automática del código a partir de pragmas, el compilador Mercurium debería poder utilizar el código añadido y las nuevas llamadas al *runtime*. Por ello, haremos una modificación manual del código generado por Mercurium adaptándolo a nuestras necesidades.

Los cambios necesarios son:

1. Las llamadas a malloc han sido cambiadas por llamadas a **nanos_FPGAMalloc.**
2. En el código de creación de la tarea, se ha pasado como tipo, el tipo FPGA correspondiente al número 3.
3. Como función **run**, se ha pasado el tipo de acelerador. Esto es necesario para que la distinción entre FPGA y SMP se haga a nivel de tipo de tarea, y la implementación *hardware* de la tarea sea tratada en el nivel de FPGAScheduler en el *runtime.*
4. Se ha añadido una llamada a **nanos_FPGAtaskwait** para esperar a la finalización de todas las tareas FPGA.

9 Análisis de rendimiento y evaluación

En este capítulo se realiza el test de funcionamiento del soporte añadido al runtime Nanos6 para la ejecución sobre aceleradores en una FPGA.

9.1 Comprobación de resultados

Para poder comprobar que la implementación funciona, se ha hecho una prueba utilizando una función Matmul de 32x32 implementada en hardware, y su homóloga en *software*.

A continuación, se muestra la función utilizada para la ejecución *software* del programa.

```
void Matmul(float *a, float *b, float *out_c)
{
  for (int i = 0; i < 32; i++)
    for (int j = 0; j < 32; j++)
    {
        float sum = out_c[i * 32 + j];
        for (int k = 0; k < 32; k++)
            sum += a[i * 32 + k] * b[k * 32 + j];
        out_c[i * 32 + j] = sum;
    }
}
```

Figura 35: Matmul serie para un bloque de 32x32

Para poder comprobar la fiabilidad del resultado, simplemente hay que hacer una comparación de las matrices de salida de la función.

Para nuestra implementación, el resultado de hacer un Matmul es correcto para la ejecución en FPGA. Por lo que cumplimos el objetivo del trabajo.

9.2 Dependencias

Una de las partes más importantes de la ejecución paralela, es el análisis de dependencias de datos.

Si una tarea necesita datos que están siendo calculados, no puede empezar hasta que estos datos no puedan ser servidos. Bajo la premisa de nuestro trabajo, en donde una tarea FPGA al finalizar su ejecución, dispone todos los datos a la memoria principal, el mecanismo de dependencias por defecto de Nanos6 es suficiente para asegurar el cumplimiento de la especificación de las dependencias. No obstante, se ha hecho una prueba para comprobar el correcto funcionamiento de las mismas.

9.3 Ejecución Matmul en FPGA

Para la ejecución del MatrixMultiply en una matriz, usaremos una estrategia llamada *tiled* Matrix Multiply, en la que dividiremos la matriz en bloques que pueden ser multiplicados independientemente.

El algoritmo se puede ver a continuación, la variable msize se refiere al tamaño de la matriz, mientras que BSIZE se refiere al tamaño del bloque (o subdivisión) que hacemos de la matriz.

```
int n = msize/BSIZE;
for (int i = 0; i < n; i += BSIZE) {
    for (int j = 0; j < n; j += BSIZE) {
        block_clear(&(c_out[i][j]), BSIZE, n);
        for (int k = 0; k< n; k+= BSIZE)
         Matmul(&(a[i][k]), &(b[k][j]),&(c_out[i][j]), BSIZE, n);
        }
}
```

Figura 36: Implementación de Tiled Matmul

Para la ejecución de este algoritmo en FPGA, hay que sintetizar como mínimo un acelerador que contenga la función Matmul, encargada de multiplicar un bloque de la matriz.

Hay que observar que al no haber dependencias de datos, si tenemos más de un acelerador potencialmente podemos reducir el tiempo de manera proporcional al número de aceleradores que tenemos.

A la hora de hacer la comprobación, hemos sintetizado un Matmul de tamaño 32x32.

En la siguiente figura se puede apreciar lo que sería una ejecución de un Matmul en dónde el tamaño del programa es ocho veces el tamaño del bloque.

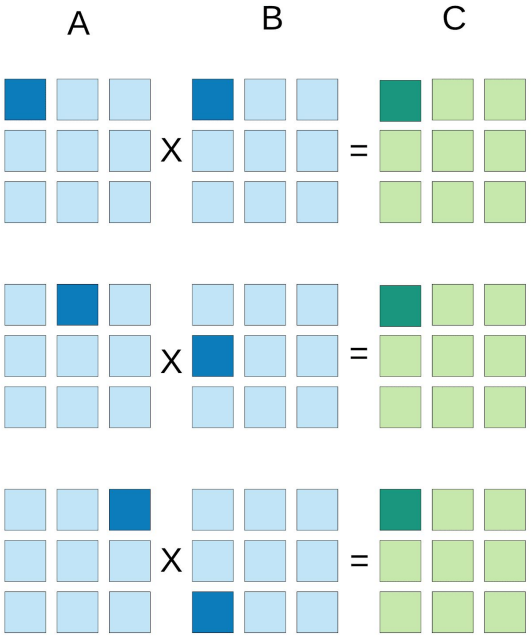

Figura 37: Ejemplo de un Tiled Matmul

Para hacer nuestra muestra de tiempo, hemos utilizado el tiempo que tarda un bloque en ser ejecutado. En la siguiente figura, se puede ver una comparativa entre Nanos5, Nanos6 y una ejecución en serie.

Ejecución	Tiempo(s)
Nanos5	0,003253
Nanos6	0,003354
Serie	0.003438

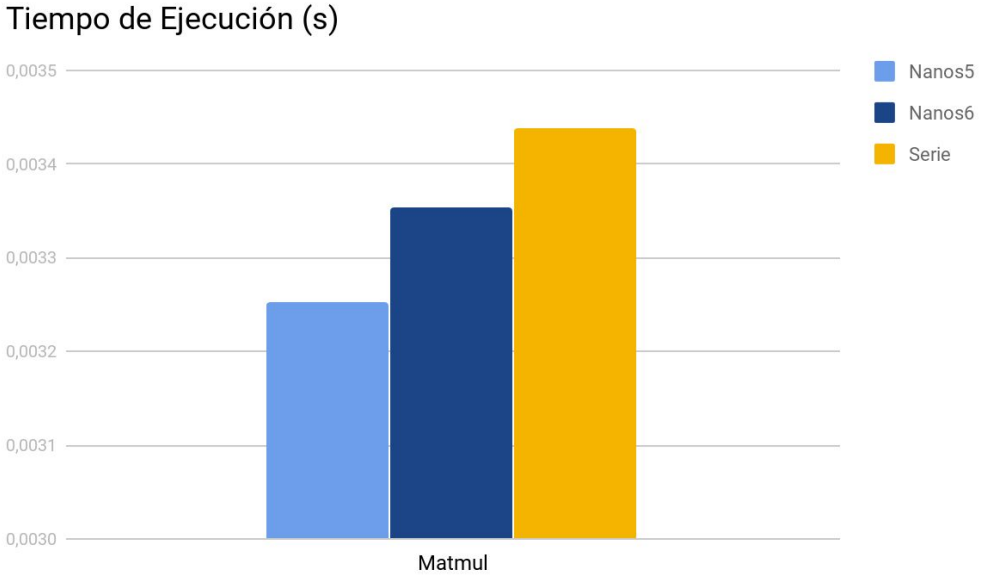

Figura 38: Tiempo de ejecución de un bloque en Nanos5, Nanos6 y ejecución serie

Como podemos apreciar, la ejecución en Nanos5 es más rápida que su contraparte en Nanos6. y ambos son más rápidos que una ejecución en serie.

Además, se ha hecho una prueba de ejecución de dos bloques a la vez, utilizando únicamente Nanos6 y ejecución serie. Esto se ha hecho para comprobar si efectivamente la implementación funciona para cuando hay más de un acelerador FPGA del mismo tipo.

Como podemos ver en el siguiente gráfico, el tiempo de ejecución en paralelo de dos bloques en aceleradores distintos concuerda con el tiempo de ejecución en uno.

Sin embargo, en una ejecución serie, el tiempo aumenta al doble.

Ejecución	**Tiempo(s)**
Nanos6	0.003329
Serie	0.006925

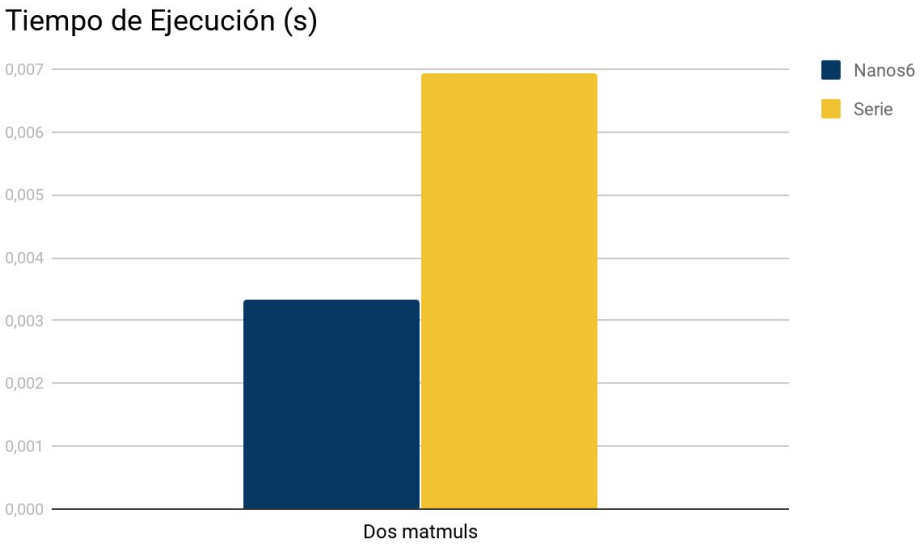

Figura 39: Tiempo de ejecución para dos aceleradores en Nanos6 y para ejecución serie.

10 Conclusiones

El objetivo principal de este trabajo es la implementación de una prueba de concepto en la que se demuestre que es posible utilizar FPGAs para acelerar aplicaciones dentro del *runtime* Nanos6.

La familiarización y instalación del entorno del modelo de programación OmpSs-2 ha sido un proceso largo que ha requerido de varias iteraciones y reuniones semanales con el equipo de devices de OmpSs-2. No obstante, este aprendizaje ha sido necesario para poder entender el proyecto y la organización del *runtime*.
Gracias a este seguimiento continuo, la implementación se ajusta más a lo que se espera del código del *runtim*e, y será más fácil en un futuro ajustarlo a la versión estable.

Con respecto del uso de FPGAs en el proyecto, se ha conseguido no solo el conocimiento necesario para utilizar estos dispositivos, sino para crear las estructuras necesarias para tratar programáticamente con las diferentes características intrínsecas del uso de aceleradores hardware.

Del mismo modo, se ha decidido soportar en nuestra versión únicamente un modelo de memoria para FPGA, en la que se considera que la placa no dispone de memoria privada, por lo que comparte espacio de direcciones con la memoria principal. Esto corresponde al modelo adoptado por la placa de desarrollo utilizada para llevar a cabo este proyecto, aunque en un futuro deberá incluir el uso de la funcionalidad para todos los modelos de memoria que quieran ser soportados.

Se ha conseguido mediante la familiarización expuesta anteriormente, crear una solución que comparte filosofía con la implementación para GPU cuya implementación está siendo llevada a cabo actualmente por el BSC, de forma que se ha procurado generalizar e integrar los paradigmas expuestos por esa implementación.

Al final, se ha conseguido un resultado satisfactorio, consiguiendo integrar en el *runtime* la funcionalidad. El testeo ha sido correcto consiguiendo tiempos parecidos, aunque ligeramente superiores a la versión anterior del *runtime*.

Este trabajo no habría sido posible sin el conocimiento previo de algunas asignaturas de la carrera, como por ejemplo PAR, en dónde nos enseñan paralelismo, PCA en dónde se toca ARM y una pincelada de FPGAs, y las diferentes asignaturas relacionadas con sistemas operativos y arquitectura de computadores.

11 Trabajo futuro

Esta prueba de concepto abre la puerta a una futura implementación en la rama principal del runtime.

En nuestra implementación, un usuario reserva memoria kernel llamando a nanos_FPGAMalloc. La dirección recibida de la llamada a esta función es la única que se puede pasar como parámetro para una tarea FPGA. Esto quiere decir que una dirección que se encuentre dentro del rango de memoria reservado no será visible.

En un trabajo futuro, se podrían buscar alternativas que solucionarían esta limitación.

Por ejemplo, una solución sería crear un directorio capaz de responder a una memoria dentro del rango.

Otra solución podría ser dejar que el usuario utilice siempre memoria del proceso, de esta forma, cuando haya que hacer una llamada a un acelerador FPGA, se puede pedir memoria kernel pinned y copiar la información necesaria ahí, y devolviendo la memoria tras la finalización del programa a la memoria del proceso en caso que sea necesario.

Estas mejoras harían que el usuario fuese totalmente agnóstico de la gestión de memoria para la ejecución de tareas FPGA.

De la misma forma, hemos supuesto memoria unificada entre FPGA y Procesador, pues es la configuración que tenía la máquina en la que hemos hecho las pruebas con el runtime. No obstante, para poder soportar muchos de los dispositivos que hay en el mercado, se necesita funcionar para más de una configuración de memoria y de modelo de ejecución.

Otra vía de investigación para trabajo futuro sería el adaptar las características de la ejecución en FPGA a una interfaz de programación en alto nivel, como por ejemplo el estándar OpenCL, de forma que creando una implementación del runtime que soporte OpenCL se estaría englobando varios dispositivos en una misma implementación.

12 Bibliografía

[1] "BSC-CNS | Barcelona Supercomputing Center - Centro Nacional de ...." https://www.bsc.es/. Se consultó el 5 de marzo 2018.

[2] "GitHub - bsc-pm/nanos6: Nanos6 is a runtime that implements the ...." 7 nov.. 2017, https://github.com/bsc-pm/nanos6. Se consultó el 5 de marzo 2018.

[3] "Release of OmpSs-2 programming model | BSC-CNS." 13 noviembre. 2017, https://www.bsc.es/news/bsc-news/release-ompss-2-programming-model. Se consultó el 5 marzo 2018.

[4] "The OmpSs Programming Model | Programming Models @ BSC." https://pm.bsc.es/ompss. Se consultó el de 5 marzo 2018.

[5] "OpenMP Application Programming Interface." 2 mar.. 2017, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. Se consultó el 5 de marzo 2018.

[6] "GitHub - bsc-pm/mcxx: Mercurium is a C/C++/Fortran source-to-source ...." https://github.com/bsc-pm/mcxx. Se consultó el 5 de marzo 2018.

[7] "What is high performance computing? - insideHPC." https://insidehpc.com/hpc-basic-training/what-is-hpc/. Se consultó el 5 de marzo 2018.

[8] "Design and Development of support for GPU Unified- UPCommons." 31 de octubre. 2017, https://upcommons.upc.edu/bitstream/handle/2117/112437/126955.pdf. Se consultó el 5 de marzo. 2018.

[9] "Intel Launches FPGA Accelerator Aimed at HPC and HPDA." 20 de diciembre. 2017, https://www.top500.org/news/intel-launches-fpga-accelerator-aimed-at-hpc-and-hpda-applications/. Se consultó el 5 marzo 2018.

[10] "OpenACC to FPGA - Future Technologies Group - Oak Ridge National ...." https://ft.ornl.gov/sites/default/files/IPDPS16_OpenACC2FPGA_PPT.pdf. Se consultó el 6 de marzo 2018.

[11] "OpenMP extensions for FPGA Accelerators." https://ce-publications.et.tudelft.nl/publications/323_openmp_extensions_for_fpga_accelerators.pdf Se consultó el 6 de marzo 2018.

[12] "Generating Hardware From OpenMP Programs - NUS Computing." https://www-new.comp.nus.edu.sg/~wongwf/papers/fpt06.pdf. Se consultó el 6 marzo de 2018.

[13] "Desarrollo en cascada - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Desarrollo_en_cascada. Se consultó el 5 de marzo 2018.

[14] "**Trello**." https://trello.com/. Se consultó el 19 de marzo 2018.

[15] "Código Deontológico - Colegio Profesional de Ingenieros" http://cpiiand.es/wordpress/download/CPIIA-Codigo_Deontologico-fCPIIA.pdf.Se consultó el 20 de marzo 2018.

[16] "Click Clean - Greenpeace." http://www.greenpeace.org/usa/global-warming/click-clean/. Se consultó el 20 de marzo 2018.